Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

On Softening OCL Invariants

Martin Gogolla[a] Antonio Vallecillo[b]

a. University of Bremen, Germany. gogolla@uni-bremen.de

b. Universidad de Málaga, Spain. av@lcc.uma.es

Abstract Invariants play a crucial role in system development. This contribution focuses on invariants in systems with so-called occurrence uncertainty, where we are interested in deciding whether a certain population (a set of instances of a class model) of the system satisfies an invariant or not, but we are unsure about the actual occurrence of the elements of that population, and also about the degree of satisfaction that is actually required for the invariant to be fulfilled. Invariants are soft in the sense that they are required to hold only for a particular, and a priori uncertain, percentage of the population. The contribution proposes a systematic approach to occurrence uncertainty and a prototypical implementation for models with uncertainty and soft invariants allowing to build system states and to make experiments with them.

Keywords UML model; Integrity constraint; OCL invariant; Uncertainty.

1 Introduction

Integrity constraints are essential elements of the specification of any information system [Oli07]. They are in charge of defining the conditions that the information system and its elements must fulfill, as imposed by the system requirements, the company business rules, the legal or fiscal regulations in force, or by the physics of the environment in which the system operates.

In UML models [Obj15], integrity constraints are normally specified by means of OCL *invariants*, which are OCL expressions of type Boolean that must be true for each instance of an entity type at any time [Obj14]. However, when dealing with models of physical systems or with those that operate in unreliable environments, these logic expressions may not be flexible and expressive enough to faithfully represent the intrinsic uncertainty of such systems.

For example, suppose that ϕ is an invariant that states that the number of instances of class X should be greater than 300. Then, if m_1 is a model with 400 instances of class X, and m_2 is a model with none, we can clearly conclude that ϕ holds for m_1 (or m_1 *satisfies* ϕ, also noted as $m_1 \models \phi$), but it does not hold for m_2. Imagine then a model that represents a Wireless Sensor Network (WSN) composed of many spatially dispersed autonomous sensors for monitoring the physical conditions of the

 In Journal of Object Technology, vol. 18, no. 2, 2019, pages 6:1–22. doi:10.5381/jot.201Y.18.2.a6

environment — like temperature, sound, pollution levels, humidity, wind, and so on. Normally these are small and inexpensive devices, whose batteries and connections are rather unreliable, and therefore there is a high uncertainty about the number of sensors that are operating at a given moment in time — because we cannot be sure that the sensor represented by a model instance is actually working, or if the links represented in the model are really active. Suppose that our system requires *around* 300 active and connected sensors to provide meaningful results. In this case, checking whether condition ϕ holds for this system is not easy, because we are uncertain about the actual number of sensors currently active and connected, and also about the degree of satisfaction required to fulfill it.

This paper proposes a softer version of OCL invariants that allows for some degree of flexibility on the constraints defined for a system when it is subject to uncertainty. In particular, we aim at answering the following research questions:

Q1. How do we *soften* (i.e., relax) existing OCL invariants that represent integrity constraints of a system to express some degree of uncertainty about their level of fulfillment?

Q2. How do we represent these soft invariants at the model level, so that existing modeling tools can properly operate with them, and modelers and other system stakeholders can reason about them?

Q3. How do we compose soft invariants using logical connectives such as $\wedge$, $\vee$ or $\neg$, to produce further soft invariants? How is uncertainty propagated?

Q4. What is the semantics of *invariant independence* in this context? Two invariants are said to be independent if none of them implies the other.

This contribution addresses two kinds of uncertainties. We first introduce the degree of satisfaction that is required for an invariant to hold, permitting invariants to be partially fulfilled—hence the name of *soft* invariants. In addition, we focus on those systems affected by *occurrence* uncertainty [BBMV18], whereby we are interested in deciding whether a given population of the system satisfies an invariant or not, but we are unsure about the actual occurrence of the elements of that population, i.e., the set of its instances.

Solving this problem is relevant in several situations. For example, in social media applications with millions of users, and running on thousands of nodes, where some of the relationships between the users are uncertain because they are derived, e.g., by recommended systems, and therefore the confidence on their existence should be taken into account when making informed decisions based upon them. Similarly, some of the invariants of these systems do not need to be satisfied by all instances, but just by a percentage of them: e.g., a constraint can be considered as fulfilled if more than 99.5% of the users satisfy it; or the system can be considered as working acceptably if less than 0.01% of the nodes are inactive. Another situation of interest happens when dealing with physical systems whose instances have an erratic or unreliable behavior, like the networks of low-cost autonomous sensors mentioned above.

Our proposal consists of softening OCL constraints by using confidence values, inequalities and thresholds. We identify several patterns that can be used to soften the OCL constraints in a systematic way, and show how standard OCL tools can be used to analyze the model, now taking uncertainty into account. In this paper we will employ the USE toolkit [GBR07] for the specification and validation of UML and OCL models.

This paper is structured in 7 sections. After this introduction, Sect. 2 briefly presents the background of our work. Then, Sect. 3 describes our proposal using an example to motivate it, and to illustrate its main concepts and mechanisms. It also presents how these concepts and mechanisms have been prototypically implemented in practice. Section 4 shows how to transform further kinds of invariants into soft ones, depending on the way they are expressed in OCL. Section 5 discusses two issues of interest in this domain: how to compose soft invariants with logical operators $\wedge$, $\vee$, $\neg$ and their combinations; and what it means for two soft invariants to be independent. Finally, Sect. 6 relates our work to other similar approaches and, Sect. 7 concludes and outlines some possible extensions and future lines of work.

2 Background

2.1 Integrity constraints and OCL invariants

Integrity constraints are rules that the system and its elements must fulfill. For example, in relational database technologies, integrity constraints are used to ensure accuracy and consistency of data. In business information systems, they define the conditions that must hold, as determined by the company policies, business regulations, or legal matters. In software applications that deal with physical systems, they are used to represent physical laws or the system environmental conditions.

In logic, integrity constraints can be defined by closed first-order formulas that the system and its elements must satisfy (i.e., they must be true) at any time. Equivalently, such constraints can be defined by predicates, called *inconsistency predicates* [Kow78], which cannot have any corresponding fact in the information base. Constraints may be atomic or compound. A constraint is compound if it can be decomposed into a conjunction of other constraints; otherwise, it is atomic.

In UML, invariant constraints are represented by means of OCL expressions that define the invariant of a type, and which must be true for all instances of that type at any time. Each invariant is always linked to the Classifier that represents that type, and that defines the scope of the invariant. Other authors propose alternative representations of invariants, e.g., by using query operations that evaluate the invariant and return a Boolean value [Oli07].

OCL invariants impose completely intransigent (*crisp*) requirements on the system: either they are satisfied or not. However, in many realistic situations, uncertainty needs to be explicitly accounted for in the invariants, in order to faithfully represent the uncertainty of the system. Otherwise, this oversimplification of the specifications would neglect some intrinsic properties of the system's nature. This is also the approach lately proposed in AI to cope with ethical decisions, where some authors suggest to let the system be explicitly unsure, the user being the one who finally decides [Eck18].

2.2 Occurrence uncertainty

Uncertainty can be defined as the quality or state that involves imperfect and/or unknown information. It applies to predictions of future events, estimations, physical measurements, or unknown properties of a system [JCG08]. We can distinguish between aleatory and epistemic uncertainty [ODR+02]. *Aleatory* uncertainty refers to the inherent variation associated with the physical system under consideration, or its

environment. In contrast, *epistemic* uncertainty refers to the potential inaccuracy in any phase of the modeling process that is due to the lack of knowledge.

The U-Model [ZSA+16, ZAY+17] identifies different kinds of uncertainties, each one requiring its own representation and operations for propagating it (measurement, occurrence, location, belief, and indeterminacy uncertainty, among others).

Among them, *Occurrence Uncertainty* (OU) is a kind of aleatory uncertainty that refers to the degree of belief that we have on the actual existence of an entity, i.e., the real entity that a model element represents. A very typical example occurs in systems whose objects represent physical events, such as those produced by (unreliable) sensors. Sometimes we may find false negatives — i.e., the event has occurred but it was lost, or the system did not record it properly. Likewise, in a social network environment, the establishment of certain derived relationships between two objects (friendship, closeness, preference) may be subject to uncertainty because the person or program estimating such a derived relationship (e.g., a recommender system) may not be fully reliable, or not completely sure about the suggestion made, and therefore it may have some associated margin of error. Here, it is interesting to distinguish between the *real* and the *model* elements: the former ones happen in reality; the latter are the ones contained in the system model, representing the real ones.

To associate occurrence uncertainty to model elements — both to objects and to relationships — we will follow [BBMV18], and use attributes that permit indicating the *confidence* (i.e, the degree of belief) that we have on their existence, by means of probabilities associated to them.

3 Expressing Soft Invariants

Fig. 1 gives a simple example for our approach of handling soft OCL invariants. It shows a crisp and a soft UML and OCL model, representing a registry of `Persons`, each one with a certain `age`, together with an integrity constraint `AllowDrive` that states that all persons in the system should be older than 18 (i.e., `p.age>=18`).

In the upper part, a conventional crisp UML model with only one OCL invariant expressing that integrity constraint, together with a system state violating the constraint, is displayed: two persons, `bob` and `ada`, are younger than required.

The lower part of Fig. 1 captures the corresponding soft model. Most notably, the soft model introduces (*a*) for the original class `Person` an additional attribute `confid`(ence) that expresses the degree of occurrence uncertainty of a respective object, and (*b*) an additional (singleton) class `Dashboard` that is responsible for expressing the degree of uncertainty (or softness) of the OCL invariant. This class holds an attribute `PersonConfidTh` for a threshold level that determines which objects are considered as relevant for the soft constraint, an attribute `AllowDriveRSL` (Required Satisfaction Level) expressing the degree of softness of the OCL invariant, and a derived attribute `AllowDriveCSL` (Current Satisfaction Level) determining the degree to which a current system state satisfies the soft invariant.

Both, confidence and satisfaction levels, are expressed in our prototypical implementation by small integers between `0` and `10`: the higher the integer, the higher the confidence; a percentage of, e.g., `7` encodes `70` percent. Although the natural choice for expressing probabilities and confidence levels would have been to use real numbers in the range [0..1], it would have prevented us to employ the USE model validator for reasoning about the softened invariants, because it relies on a constraint solver and hence it uses integer arithmetic. For this reason, we have preferred to lose precision,

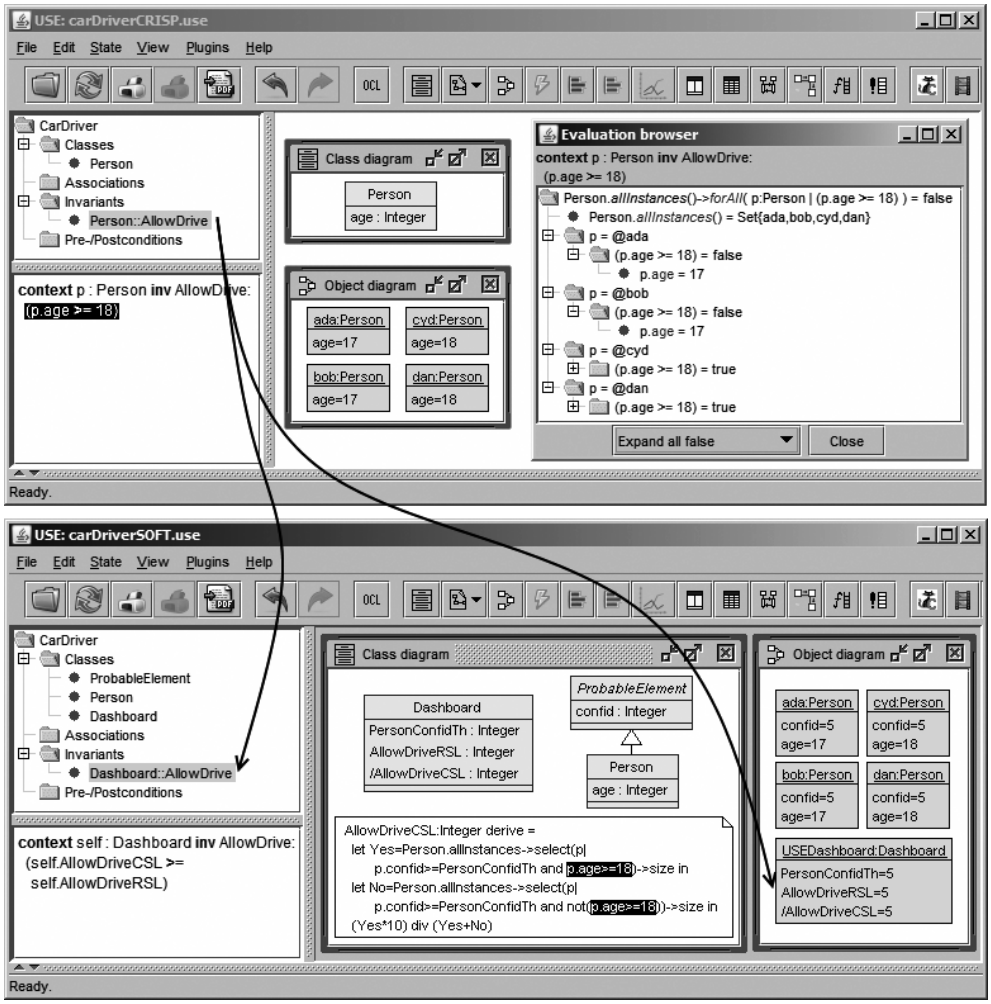

Figure 1 – Crisp and soft version of CarDriver example model.

even when some of the OCL expressions become more complex and therefore less readable; but in return we can efficiently perform different types of analyses on the invariants (see Sect. 5). In any case, if the model validator is not required in any particular application, real numbers representing probabilities could be used instead, since the proposed approach can be equally applied.

The original invariant is expressed by requiring that the CSL is greater or equal to the RSL; in this way, the original invariant becomes part of the CSL derivation rule. The original failing system state now becomes a valid state, because `AllowDriveRSL=5` states that only 50 percent of the objects (that have a threshold greater or equal to the `Person` threshold) have to fulfill the age requirement.

This is the standard method in which OCL invariants are *softened* in our proposal:

(*a*) Decide which are the entities that can be subject to occurrence uncertainty, and add to them an attribute `confid` with a probability; this is normally done by extending from abstract class `ProbableElement` [BBMV18].

(*b*) For each invariant `<Inv>`, associated to class `<C>`, create three attributes.

- The first one, `<C>ConfidTh:Integer`, determines the minimum level of confidence required for an object of class `<C>` to be considered as existing.
- The second attribute, `<Inv>CSL:Integer`, is derived. It defines the *current* satisfaction level for the invariant.
- The last attribute, `<Inv>RSL:Integer`, is defined by the user and determines the minimum *required* satisfaction level for the invariant to hold, according to the user's requirements or judgments.

(*c*) A new class `Dahsboard` contains all these newly defined attributes for each invariant. Changing the values of these thresholds will allow modelers to easily simulate different uncertainty levels and degrees of satisfactions for the invariants.

(*d*) Finally, we reformulate each invariant `<Inv>` in terms of the corresponding expression, `<Inv>CSL >= <Inv>RSL`, that states that the current satisfaction level for the invariant should be above the required one.

Note how the model in the lower part of Fig. 1 has been developed using the steps defined in this process.

Fig. 2 shows how the valid soft constraint and the system state from Fig. 1, which is repeated in the centre, could be violated and lead to 4 different failing system states:

(1) Changing the class threshold. A first option for achieving an invalid system state is to increase the confidence threshold for the objects (top-left). Raising it from `5` to `6` disregards all `4` objects as being relevant, and reduces the `CSL` to `0`.

(2) Changing the required satisfaction level. A second option (top-right) is to raise the required satisfaction level. Increasing `RSL` to `6` means that at least `60` percent of the objects over the threshold have to satisfy the age condition `age>=18`. This leads to the violation, since exactly `50` percent satisfy it.

(3) Changing an object threshold. A third option (bottom-left) is to decrease the confidence for one object satisfying the age condition. This means the `CSL` becomes only `2`, which represents the percentage `25` and that is below the required satisfaction level `5`.

(4) Changing an object attribute value. A last option (bottom-right) is to lower the `age` attribute value for one object previously satisfying the age condition. Again this means that the `CSL` is lowered to `2`, making the constraint fail.

In this way, we can use the `Dashboard` to simulate how changes in these different parameters affect the invariants. The higher the thresholds and the required satisfaction levels, the more stringent the invariants. And vice-versa: lowering the thresholds and the degree of satisfaction required for an invariant, the more permissive they become.

This is graphically displayed in Fig. 3 that explains in form of a table the classification and distinction between crisp and soft invariants which is prominently determined by the dashboard settings. In the left, vertical dimension, the confidence specifications from the model are represented, whereas in the bottom, horizontal dimension, the degree of constraint uncertainty is displayed. A completely crisp invariant would refer to model elements with confidence `10` and a required satisfaction level of `10`. A completely soft invariant would refer to completely uncertain model elements with

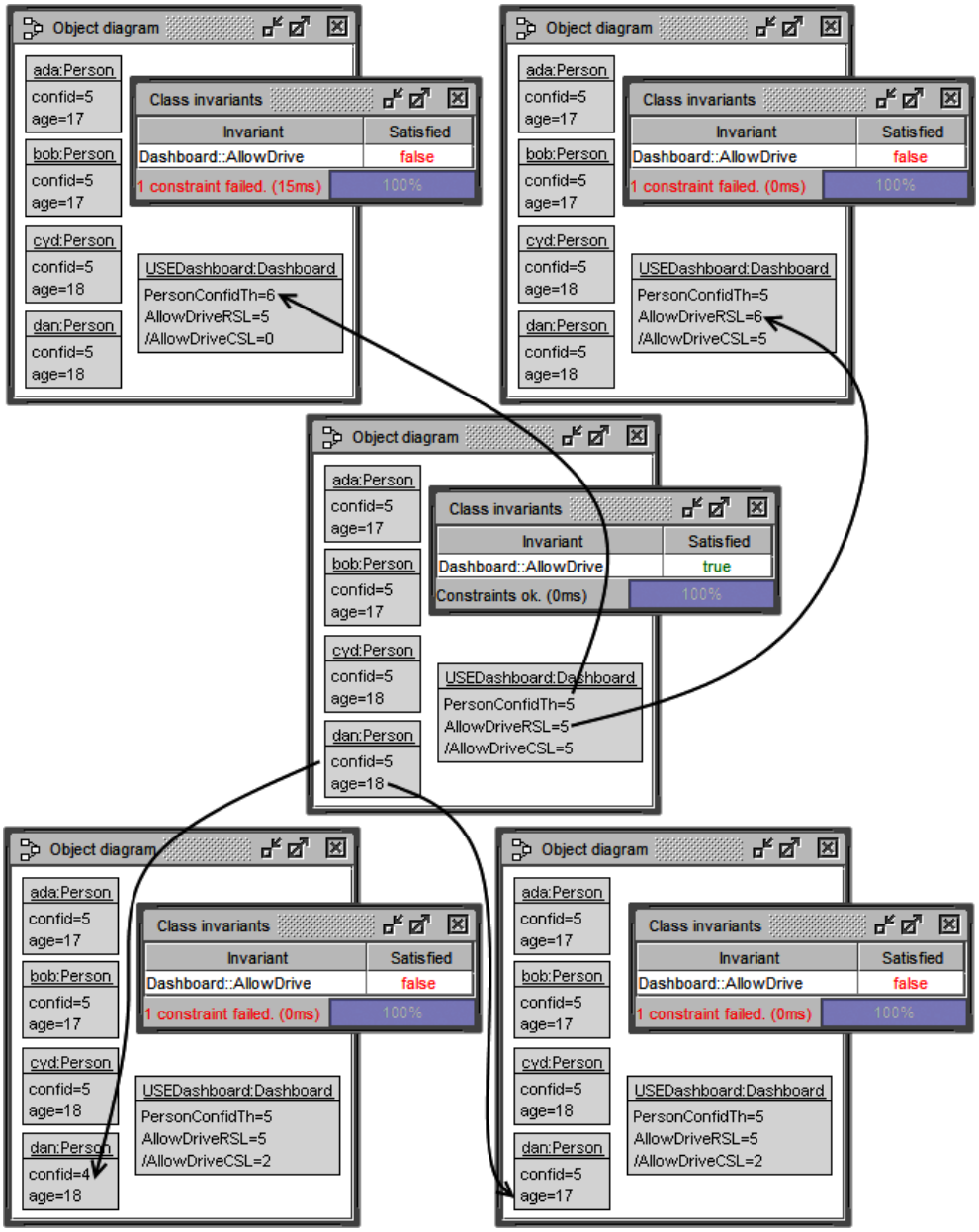

Figure 2 – Four options for introduction of failing object diagrams.

confidence 0 and a required satisfaction level of 0. Changing the values of the confidence threshold and the required satisfaction level of an invariant enables interesting simulations that permit representing different degrees of uncertainty: both on the occurrence of the instances and on the degree of satisfaction required for the invariant to hold. For example, we can specify that we only need an invariant to be fulfilled in 90% of the cases, and that we want to consider only those instances for which we are more than 99% confidence of their occurrence.

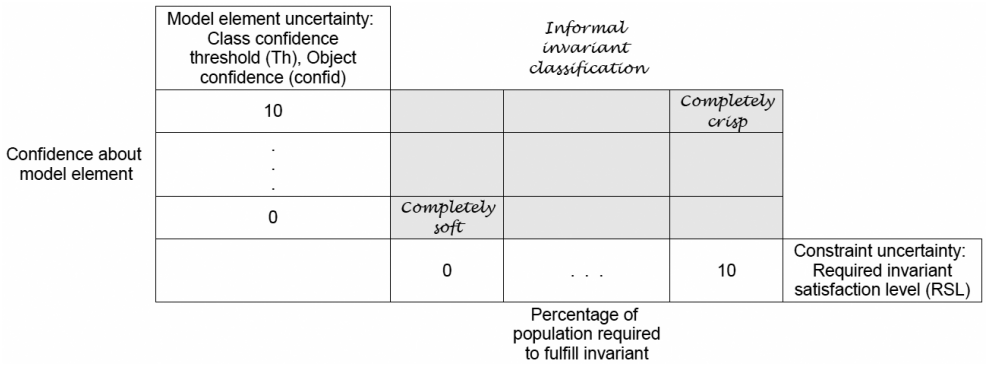

Figure 3 – Using the elements' confidence thresholds and the required satisfaction level of the soft invariants to obtain different grades of flexibility/stiffness in invariants.

4 Invariant Patterns

Once we know how to associate degrees of satisfaction to the invariants, and uncertainty to the model elements restricted by them, we are faced with the problem of how to define the derivation expression that calculates the *current satisfaction level* (CSL) for a given OCL invariant, which is precisely how to write the expression that softens the invariant. In this section we see how the formula of such a derived OCL expression will depend on the kind of invariant, since it is based on the structure and the type of constraint they impose on the model.

In the previous example, the relaxed (i.e., softened) invariant was about requiring that an attribute of the instances of a given class should fulfill a certain constraint defined by an OCL expression (`age>=18`). Here we will also analyze other invariant *patterns*, namely those that impose that a certain object, or a link between two instances, exists; that certain sets of instances of a model should have the same size; or that the set of instances that satisfies a given condition should be empty (or not). We will introduce these kinds of constraints and how to relax them using examples.

4.1 The LOTR Wars example

This example illustrates how invariants of two different types can be *softened*. The first one represents the kind of invariant that asks two sets to be of the same size, i.e., `SetA->size()=SetB->size()`. The second one requests a set not to be empty, i.e., `SetC->notEmpty()`.

Suppose the model described in Fig. 4, which specifies a typical scenario in the Lord of The Rings (LOTR) saga. There are `Battles` between `Allies` and `Enemies`. Suppose two kinds of allies, `Dwarfs` and `Elves`, and two kinds of enemies, `Orcs` and `Spiders`.

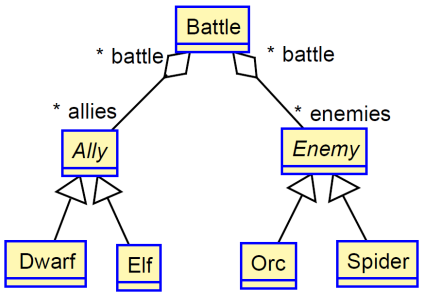

Figure 4 – The LOTR Wars metamodel.

The system defines three constraints. The first one (`FairBattle`) requires that all battles should be fair, i.e., the number of enemies should be on par with the numer

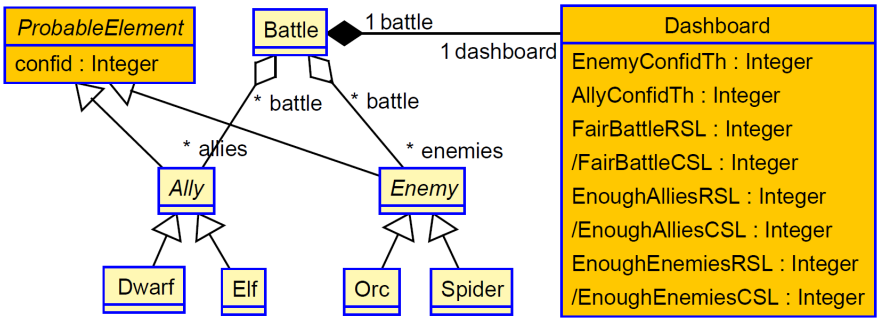

Figure 5 – The LOTR Wars metamodel, enriched with confidence and a dashboard.

of allies. The other two invariants (`EnoughAllies` and `EnoughEnemies`) state that the sets of allies and enemies of a battle can not be empty. These invariants can be specified in OCL as follows.

```
context Battle inv FairBattle: self.enemies->size = self.allies->size
context Battle inv EnoughAllies: self.allies->notEmpty
context Battle inv EnoughEnemies: self.enemies->notEmpty
```

In this example, taking into account the information about the occurrence uncertainty of each element is very relevant. From the distance, due to the smoke of the battle, we can not be completely sure if the objects that we see are enemies, allies, or simply rocks. This is why we need to assign a degree of confidence to the model instances that represent allies and enemies.

Fig. 5 shows the LOTR Wars metamodel enriched with the information required to describe the confidence of the elements that are subject to occurrence uncertainty (abstract class `ProbableElement`) and the `Dashboard`.

The `Dashboard` defines the attributes for regulating the level of flexibility of each invariant. Note that in this case we are associating a dashboard to each `Battle` object, instead of having only one for the complete system. The dashboard defines the thresholds that determine the minimum level of confidence to consider an enemy or an ally as such (`EnemyConfigTh` and `AllyConfidTh`), as well as the required (RSL) and current satisfaction (CSL) levels for the three original OCL invariants.

The required satisfaction level (RSL) is an attribute that the modeler can set (and change) during the system execution. In contrast, the current satisfaction level (CSL) is a derived attribute that computes the percentage of objects that currently satisfy the invariant. In the LOTR Wars example, invariant `FairBattle` can be relaxed by requiring that difference between the sizes of the sets of enemies and allies is a small percentage of the overall size of the set of enemies and allies together. With this idea in mind, the corresponding attributes for invariant `FairBattle` are defined as follows.

```
FairBattleRSL : Integer -- Required satisfaction level (user defined)
FairBattleCSL : Integer derive = -- Current satisfaction level
  let YesE=battle.enemies->select(e|e.confid>=EnemyConfidTh)->size in -- # real enemies
  let YesA=battle.allies->select(a|a.confid>=AllyConfidTh)->size in -- # real allies
  let diff = if YesA>=YesE then YesA-YesE else YesE-YesA endif in
    10 - ((diff * 10) div (YesE + YesA)) -- i.e., 1-(YesA-YesE).abs()/(YesE+YesA)
```

As described earlier, we use integer numbers between 0 and 10 to represent probabilities, so that the USE model validator can employ integer arithmetic. In this case, the current satisfaction level (`FairBattleCSL`) is computed as the difference (in

absolute value) between *real* enemies and allies (as determined by the corresponding thresholds), divided by the total number of real enemies and allies. With this, the `FairBattle` invariant can be simply reformulated as a comparison that checks if the current satisfaction level (as automatically derived) is above the required one (as stated by the user):

```
context b:Battle inv FairBattle:
  b.dashboard.FairBattleCSL >= b.dashboard.FairBattleRSL
```

Using this approach, we can allow for a small percentage of variation, e.g., a 5 or a 10%, between the two sides of the battle. The value of such a allowed percentage is precisely what attribute `FairBattleRSL` expresses.

The other two invariants required the sets of enemies and allies not to be empty. We will show here how to soften one of them (`EnoughAllies`), the other is similar.

Again, we need to define two attributes in the dashboard, one for the required satisfaction level and another for the current one:

```
EnoughAlliesRSL : Integer
EnoughAlliesCSL : Integer derive =
  let AllA=battle.allies->size in -- all allies
  let YesA=battle.allies->select(a|a.confid>=AllyConfidTh)->size in -- real allies
  (YesA * 10) div AllA
```

In this case, the current satisfaction level represents the ratio of *real* allies in the battle against the ones declared as such in the model.

With this, the expression of the softened invariant becomes a comparison between the required and the current satisfaction levels:

```
context b:Battle inv EnoughAllies:
  b.dashboard.EnoughAlliesCSL >= b.dashboard.EnoughAlliesRSL
```

Now, to simulate the system, we need to decide first the lower and upper limits on the number of instances of each type, and determine the thresholds and satisfaction levels that we want to achieve. We can ask the USE model validator to find a model that satisfies these restrictions. For example, we can provide the USE model validator with a set of restrictions that limits the number of allies and enemies to 3 orcs, 2 spiders, 4 elfs and 1 dawrf, and that sets to 0.8 all thresholds and satisfaction levels:

```
Orc_min = 3
Orc_max = 3
Spider_min = 2
Spider_max = 2
...
Dashboard_EnoughAlliesRSL = Set{ 8 }
Dashboard_EnoughEnemiesRSL = Set{ 8 }
Dashboard_FairBattleRSL = Set{ 8 }
Dashboard_AllyConfidTh = Set{ 8 }
Dashboard_EnemyConfidTh = Set{ 8 }
```

The USE model validator tries to generate a system state of instances and links that respects these restrictions, and also satisfies all the system OCL invariants. Fig. 6 shows the system state found in this case, as well as the invariants. Using this system state as starting point, we can now simulate the effect of setting different thresholds or required satisfaction levels, and how the evaluation of the system invariants change, as we did for the `CarDriver` example.

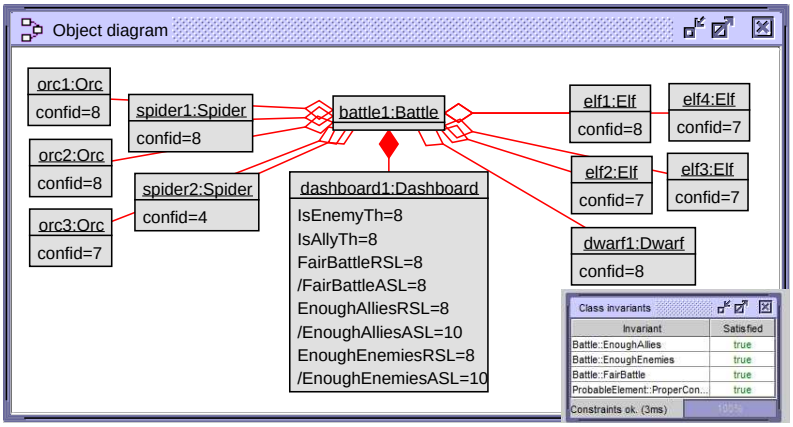

Figure 6 – System state generated by the USE model validator for the LOTR example.

4.2 The Likes-Follows example

In this section we will discuss how to soften invariants that require certain objects to exist, or not, in the system.

Fig. 7 represents a system composed of `Users` and `Items`. Users may follow other users, and may like items. Relations `Likes` and `Follows` are represented as classes, too. It is common in recommender systems to create these two types of relationships (likes and follows) based on the current state of the users. For example, if more than N followers of a user `u` like a particular item `itm`, the system creates a `Likes` relationship between `u` and `itm`. Probabilities are normally assigned to those auto-generated relationships. They define the degree of confidence that the system assigns to them depending, e.g., on the number N of followers that also like the item, on the confidence of these `Likes` relationships (since they can be derived, too), and on the confidence the system has on the derivation rule.

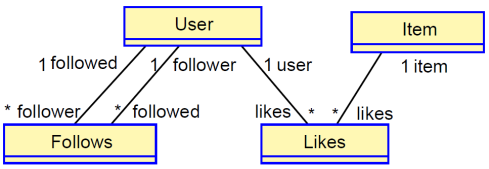

Figure 7 – The Likes-Follows metamodel.

The system defines four constraints. The first one (`notFollowingSelf`) requires that users do not follow themselves. The next two (`noLikesDups` and `noFollowsDups`) forbid the occurrence of duplicate relationships between two users, or a user and an item. The last invariant (`allLiked`) does not permit an item to exist without any user liking it. In a real system, however, we should allow things to be more flexible — especially if we introduce the use of confidence level assigned to relationships. The crisp version of these OCL invariants in shown below; we have also added, as comments, how each individual invariant can be relaxed.

```
context u:User inv notFollowingSelf: -- SOFT: few following with low confidence
    u.follower->forAll(fR | fR.follower<>u)
context l1:Likes inv noLikesDups: -- SOFT: few dups with low confidence
    not Likes.allInstances->exists(l2 |
      not (l1<>l2 implies (l1.user<>l2.user or l1.item<>l2.item)))
context f1:Follows inv noFollowsDups: -- SOFT: few dups with low confidence
  Follows.allInstances->forAll(f2 |
    f1<>f2 implies (f1.follower<>f2.follower or f1.followed<>f2.followed))
context i:Item inv allLiked: -- SOFT: few not liked with low confidence
  i.likes->notEmpty
```

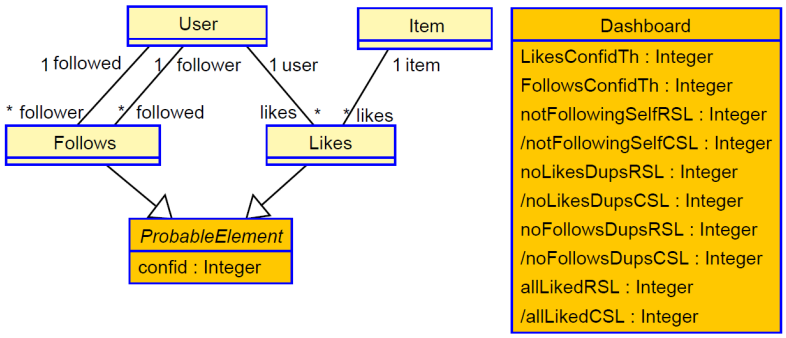

Figure 8 – The Likes-Follows metamodel, enriched with confidence and a dashboard.

Fig. 8 shows the Likes-Follows metamodel enriched with the information required to describe the confidence of the elements that are subject to occurrence uncertainty (abstract class `ProbableElement`) and the `Dashboard`.

In this case we are associating a dashboard to the complete system. It defines the attributes for regulating the level of flexibility of each invariant. These attributes specify the thresholds that determine the minimum level of confidence to consider a `Likes` or `Follows` relationship to be significant (`LikesConfidTh` and `FollowsConfidTh`), as well as the required and current satisfaction levels for the four (crisp) OCL invariants.

In the Likes-Follows example, the `notFollowing` invariant can be relaxed by allowing a small percentage of users to follow themselves. That percentage is precisely what required satisfaction level `notFollowingSelfRSL` determines. Then, the corresponding attributes for the `notFollowing` invariant can be defined as follows.

```
notFollowingSelfRSL : Integer -- required satisfaction level (user defined)
notFollowingSelfCSL : Integer derive = -- current satisfaction level
  let Yes = User.allInstances->select(u | u.follower->select(fR |
              fR.confidence>=FollowsConfidTh and fR.follower=u)->isEmpty)->size in
  let No = User.allInstances->select(u | u.follower->select(fR |
              fR.confidence>=FollowsConfidTh and fR.follower=u)->notEmpty)->size in
  (Yes*10) div (Yes+No)
```

Here, the current satisfaction level (`notFollowingSelfCSL`) is computed as the ratio, expressed in integer arithmetic, between the actual users that fulfill the constraint, and the total number of real users. Then, the `notFollowingSelf` invariant can be simply reformulated as a comparison between the current satisfaction level (as automatically derived) and the required one (as stated by the user):

```
context Dashboard inv notFollowingSelf: notFollowingSelfCSL >= notFollowingSelfRSL
```

The following two invariants banned the existence of duplicated instances of the objects that reify the two relationships, `Follows` and `Likes`. The dashboard defines again the corresponding attributes, to set the required and current satisfaction levels.

```
noLikesDupsRSL : Integer -- required satisfaction level
noLikesDupsCSL : Integer derive = -- current satisfaction level
  let Yes = Likes.allInstances->select(l1 | l1.confidence>=LikesConfidTh and
              Likes.allInstances->select(l2 | l2.confidence>=LikesConfidTh and
                l1<>l2 and l1.user=l2.user and l1.item=l2.item)->isEmpty)->size in
  let No = Likes.allInstances->select(l1 | l1.confidence>=LikesConfidTh and
             Likes.allInstances->select(l2 | l2.confidence>=LikesConfidTh and
               l1<>l2 and l1.user=l2.user and l1.item=l2.item)->notEmpty)->size in
  if Yes+No=0 then 0 else (Yes*10) div (Yes+No) endif
```

The current satisfaction level represents the ratio of duplicates against the total number of instances of that relationship, considering only those instances whose confidence is above threshold `LikesConfidTh`. The expression of the softened invariant becomes a comparison between the required and current satisfaction levels:

```
context Dashboard inv noLikesDups: noLikesDupsCSL >= noLikesDupsRSL
```

The final invariant banned items that no single user liked. This invariant can be softened by requesting that the percentage of items which are either not liked by any users, or liked with very little confidence, is small — or, equivalently, that the percentage of items which are either liked by some users, or liked with enough confidence, is above the the required satisfaction level:

```
allLikedRSL : Integer -- required satisfaction level
allLikedCSL : Integer derive = -- current satisfaction level
  let Yes = Item.allInstances->select(i|i.likes->notEmpty)->
              select(i|i.likes->exists(l|l.confid>=LikesConfidTh))->size in
  let No = Item.allInstances->select(i|i.likes->isEmpty or
           (i.likes->notEmpty and i.likes->forAll(l|l.confid>=LikesConfidTh)))->size in
  if Yes+No=0 then 0 else (Yes*10) div (Yes+No) endif
```

The corresponding invariant is specified as follows:

```
context Dashboard inv allLiked: allLikedCSL >= allLikedRSL
```

4.3 OCL invariant patterns

So far we have seen how various types of OCL invariants can be relaxed, and how their soft versions are produced. These types (or *patterns*) of OCL expressions are listed below, and summarized in Table 1:

- If an invariant I_1 requests that the sizes of two sets are the same, or that one is larger or equal than the other, we can relax it by requesting that the difference between these sizes is below a percentage of the sum of the sizes of the two elements, considering only the elements above the confidence thresholds defined by the user for the invariant.
- If an invariant I_2 requires that there should be at least one element of a given type X, we can soften it by considering the elements whose confidence is above the threshold defined by the user, dividing it by the total number of instances of X in order to obtain a ratio.
- If an invariant I_3 requires that a set of elements that fulfill a property P must not be empty, we can soften it by considering: (a) the elements whose confidence is above the threshold defined by the user and that fulfill P; (b) the elements whose confidence is above the threshold defined by the user and that do not fulfill P, and calculate the current satisfaction level of I_3 as the proportion (a)/(a)+(b).

To soften compound invariants, obtained by the combination of other invariants with logical operators $\neg$, $\wedge$ and $\vee$, we compose their current satisfaction levels. Table 1 shows the derivation expression for them, assuming the invariants to combine are independent. The explanations for these expressions are detailed in the next section.

Table 1 – Softening OCL invariants.

Crisp invariant pattern	Corresponding soft invariant

`Invariant1 ≡`
`X->size = Y->size` or
`X->size >= Y->size`
	`let a=X->select(x|x.confid>=XConfidTh)->size in`
	`let b=Y->select(y|y.confid>=YConfidTh)->size in`
	`let Invariant1CSL = if (a+b)=0 then 0`
	`else (1-((a-b).abs()/(a+b))) endif in`
	`Invariant1CSL >= Invariant1RSL`

`Invariant2 ≡`
`X.allInstances->`
`notEmpty()`
	`let a=X.allInstances->select(x|x.confid>=XConfidTh)->size in`
	`let b=X.allInstances->size in`
	`let Invariant2CSL = if b=0 then 0 else (a/b) endif in`
	`Invariant2CSL >= Invariant2RSL`

`Invariant3 ≡`
`X->select(x|P(x))->`
`size()>0`
	`let a=X->select(x|x.confid>=XConfidTh and P(x))->size in`
	`let b=X->select(x|x.confid>=XConfidTh and not P(x))->size in`
	`let Invariant3CSL = if (a+b)=0 then 0 else a/(a+b) endif in`
	`Invariant3CSL >= Invariant3RSL`

`Invariant4 ≡` `not (OtherInvariant)`	`let Invariant4CSL = 1.0 - OtherInvariantCSL in` `Invariant4CSL >= Invariant4RSL`
`Invariant5 ≡` `InvX and InvY`	`let Invariant5CSL = InvXCSL * InvYCSL in` `Invariant5CSL >= Invariant5RSL`
`Invariant6 ≡` `InvX or InvY`	`let Invariant6CSL = InvXCSL + InvYCSL - InvXCSL*InvYCSL in` `Invariant6CSL >= Invariant6RSL`

5 Operating with Compound Soft Invariants

5.1 Combining soft invariants with logical operators

This section discusses two issues: how to compose soft invariants with logical operators $\wedge$, $\vee$, $\neg$ and their combinations; and what does it mean for two invariants to be independent. For this, we need to formulate the problem in different terms.

In logic, an integrity constraint can be defined in terms of a closed first-order formula that the information base must satisfy, i.e., it must be true. Then, if $\mathcal{M}$ represents the set of models, $\mathcal{LP}$ the set of logic predicates, and $\mathbb{B}$ the Booleans, we talk about the *satisfaction* relationship $\models : \mathcal{M} \times \mathcal{LP} \to \mathbb{B}$ that, given a model m and a logic predicate ϕ that represents an invariant, returns whether model m satisfies invariant ϕ or not (we will also note this as $m \models \phi$).

To obtain a softer version of an OCL invariant and of the satisfaction relationship "$\models$", instead of returning a Boolean value, we extend it to return a real number in the range [0..1] that represents how much the model satisfies the invariant, i.e., what we have called its *current satisfaction level*.[1] This new relationship is called *uncertain satisfaction*, $\models\!\!\!\sim\ : \mathcal{M} \times \mathcal{LP} \to [0..1]$.

It can be considered as an extension of relationship $\models$, because given a model m, and an invariant ϕ, we always have that $(m \models\!\!\!\sim \phi) = 1 \iff (m \models \phi)$.

This extension is similar to the one that lifts the `Boolean` datatype to `UBoolean`, as described in [BBMV18], and where Boolean values are extended to pairs of the form $(true, c)$, with c a real number in the range [0..1] that represents the confidence that we have on the fact that the Boolean value is true. With this, $false$ is lifted to $(true, 0)$ and $true$ is lifted to $(true, 1)$.

The algebra of operations defined for `UBoolean` can be used here and, if ϕ_1 and ϕ_2

[1] Although in the preceding sections we represented probabilities and confidence levels by means of small integer numbers for technical reasons (namely, in order to be able to employ the USE Model Validator), in this more theoretical section we have preferred to use their more natural representation as real numbers in the range [0..1] instead.

are *independent invariants* (see Sect. 5.2), then the following expressions hold:

$$m \trianglerighteq (\neg\phi_1) = 1.0 - m \trianglerighteq \phi_1 \quad (1)$$

$$m \trianglerighteq (\phi_1 \wedge \phi_2) = m \trianglerighteq \phi_1 * m \trianglerighteq \phi_2 \quad (2)$$

$$m \trianglerighteq (\phi_1 \vee \phi_2) = m \trianglerighteq \phi_1 + m \trianglerighteq \phi_2 - (m \trianglerighteq \phi_1 * m \trianglerighteq \phi_2) \quad (3)$$

Note that we are using here probability theory [dF17], and not *fuzzy logic* [Zim01]: although both approaches deal with states of uncertainty, and represent degrees of subjective belief, fuzzy set theory uses the concept of fuzzy set membership, whereby an element can belong to different sets of one partition of the whole space. In contrast, probability theory uses the concept of subjective probability, i.e., the likelihood of an event or condition to belong to each set of the partition, assuming it can only belong to one set [Kos90, DP93]. These expressions justify the equations used in Table 1.

Moreover, notice that the value of $m \trianglerighteq \phi$ coincides with the *current satisfaction level* of model m with respect to invariant ϕ, and which is defined by the derived attribute `<φ>CSL` stored in the `Dashboard` of our prototypical implementation.

5.2 Invariant independence

Traditionally, two invariants are said to be *independent* if none of them implies the other. Conversely, they are dependent if every object model that satisfies one of them, satisfies the other. This definition can be extended to sets of invariants, when no invariant from the set is an *implication* from the rest [GHD18].

In mathematical terms, let us suppose two invariants ϕ_1 and ϕ_2 that specify constraints on UML models defined by metamodel $\mathcal{M}$. Let m, m_1, m_2 be object models of $\mathcal{M}$. Traditionally, we say that ϕ_2 is ***dependent*** on ϕ_1, or that ϕ_2 is an *implication* from ϕ_1 (also noted as $\phi_1 \Rightarrow \phi_2$), iff every model that satisfies ϕ_1 also satisfies ϕ_2:

$$\forall m \in \mathcal{M} \bullet (m \models \phi_1 \Rightarrow m \models \phi_2) \quad (4)$$

Contrarily, ϕ_1 and ϕ_2 are ***independent*** iff:

$$\exists\, m_1, m_2 \in \mathcal{M} \bullet (m_1 \models \phi_1 \wedge m_1 \not\models \phi_2) \wedge (m_2 \models \phi_2 \wedge m_2 \not\models \phi_1) \quad (5)$$

In our context, we need to extend these definitions for the *uncertain satisfaction* relationship $\trianglerighteq$, that returns a real number in the range [0..1] instead of a Boolean value. In this case, the definition of dependence can be easily lifted using the natural order defined for real numbers, and then define that ϕ_2 is ***dependent*** on ϕ_1 (also noted as $\phi_1 \Rightarrow \phi_2$) iff:

$$\forall m \in \mathcal{M} \bullet (m \trianglerighteq \phi_1) \leqslant (m \trianglerighteq \phi_2) \quad (6)$$

In other words, dependency holds, if the satisfaction levels of all models w.r.t. ϕ_1 is less or equal than their satisfaction level w.r.t. ϕ_2. To improve the readability of the formulas, we will also use a shorthand notation for $m \trianglerighteq \phi$, and write it as $\phi(m)$. With this, the previous expression can be written as follows:

$$\forall m \in \mathcal{M} \bullet \phi_1(m) \leqslant \phi_2(m) \quad (7)$$

We also say that two invariants ϕ_1 and ϕ_2 are ***independent*** iff:

$$\exists\, m_1, m_2 \in \mathcal{M} \bullet \ \phi_1(m_1) > \phi_2(m_2) \ \wedge \ \phi_1(m_2) > \phi_2(m_1) \quad (8)$$

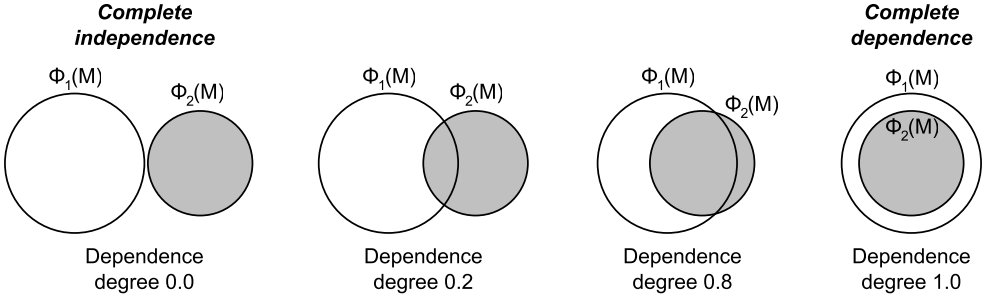

Figure 9 – Degrees of independence between invariants.

Note how this condition naturally generalizes the condition for the traditional notion of independence represented by equation (5).

However, in this context we need to express that independence is no longer a boolean (i.e., binary) property, but a continuous one (see Fig. 9), and thus we need to define a *degree* of independence between two invariants ϕ_1 and ϕ_2. For this, we need to notice first that the following condition always holds, for all models of $\mathcal{M}$:

$$\phi_1(m) * \phi_2(m) \leqslant (\phi_1 \wedge \phi_2)(m) \leqslant \min\{\phi_1(m), \phi_2(m)\} \tag{9}$$

When the invariants are dependent, we get that $(\phi_1 \wedge \phi_2)(m) = \min\{\phi_1(m), \phi_2(m)\}$, and when they are *completely* independent, we know by equation (2) that $(\phi_1 \wedge \phi_2)(m) = \phi_1(m) * \phi_2(m)$.

Then, we can define the degree of independence between two invariants ϕ_1 and ϕ_2, as a real number in the range [0..1], as follows:

$$i(\phi_1, \phi_2) = 1 - \frac{\#\{m \in \mathcal{M} \bullet (\phi_1 \wedge \phi_2)(m) > 0\}}{\#\{m \in \mathcal{M} \bullet (\phi_1 \vee \phi_2)(m) > 0\}} \tag{10}$$

In that equation, the hash sign (#) denotes the size of the set, which is defined using a set comprehension expression. The problem with this equation is that it is difficult to use in practice, because the number of models in these sets can be unbounded. What can be interesting is to concentrate on the extreme cases of complete independence and complete dependence.

Complete dependence was already defined by equation (7). Now, we can define that two invariants ϕ_1 and ϕ_2 are ***completely independent*** iff:

$$\forall\ m \in \mathcal{M} \bullet (\phi_1 \wedge \phi_2)(m) = \phi_1(m) * \phi_2(m) \tag{11}$$

Once we have all these theoretical definitions, let us show how we can define practical tests for them, employing the USE model validator.

- Invariant **independence** can be checked by asking the model validator to look for a model m_1 such that $\phi_1(m_1) > \phi_2(m_1)$ and another model m_2 such that $\phi_2(m_2) > \phi_1(m_2)$.

  For instance, in the LOTR example, we selected the `EnoughAllies` and `FairBattle` invariants, and asked the model validator to look first for one model m_1 that satisfied the first invariant and not the second one, and for another model m_2 that satisfied the second invariant and not the first one. We used the

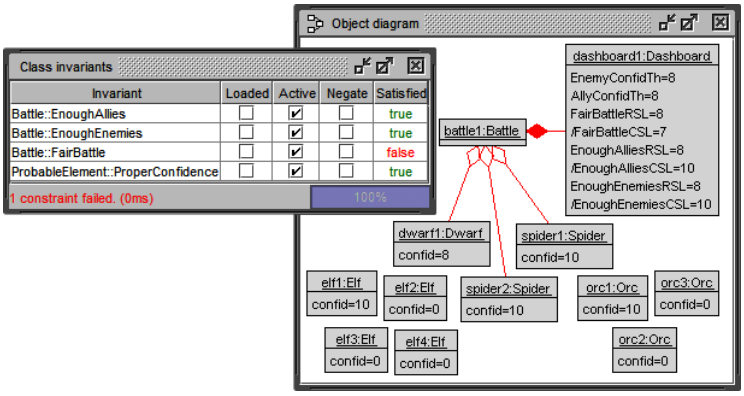

Figure 10 – First found system state for showing invariant independence.

model validator invariant negation feature to easily achieve this. The model validator successfully found a model m_1 composed of 2 allies and 1 enemy whose confidence was above the threshold (hence satisfying the request; see Fig. 10), and another model m_2 with 1 enemy and 2 allies; from the 2 allies, one had a confidence above the threshold, and the other didn't. Therefore, the model validator was able to find the models that show that these two invariants are independent.

- Invariant **dependence** can be checked by letting the model validator look for a model m such that $\phi_2(m) < \phi_1(m)$.

  In order to have two dependent invariants, we developed a new one, `EnoughAlliesANDFairBattle`, which is basically the conjunction of the two invariants `EnoughAllies` and `FairBattle`. We wanted to use the fact that for any two invariants ϕ_1 and ϕ_2, we know that $\phi_1 \wedge \phi_2 \Rightarrow \phi_1$. We then wanted to check that `EnoughAlliesANDFairBattle` is dependent on `EnoughAllies`. For this, we asked the model validator to find a model m for which the current satisfaction level `EnoughAlliesCSL` (which corresponds to $m \models$ `EnoughAllies`) is greater than `EnoughAlliesANDFairBattleCSL`. The model validator was unable to find such a model. Of course, this result is relative to the search space used, as defined by the developer when specifying the context of the search — by means of the `properties` file of the model validator. Similarly, invariants `EnoughAllies`, `EnoughEnemies` and `FairBattle` are not independent invariants, and in fact it can be checked that `EnoughAllies` $\wedge$ `FairBattle` implies `EnoughEnemies`, e.g., by using the `-invIndep` option of the model validator [GHD18].

- Invariant **complete independence** can be checked by asking the model validator to look for a model m such that $(\phi_1 \wedge \phi_2)(m) \neq \phi_1(m) * \phi_2(m)$.

  In this case we also used the invariant `EnoughAlliesANDFairBattle`, defined as the conjunction of `EnoughAllies` and `FairBattle`, and asked the model validator to look for a model for which the dashboard attribute `EnoughAlliesANDFairBattleCSL` did not coincide with the product `EnoughAlliesCSL * FairBattleCSL`. The search was unsatisfiable, confirming that the two invariants `EnoughAllies` and `FairBattle` are completely independent.

With all this, we have been able to extend the concept of invariant independence,

replacing a *binary* definition by a *softer* version that admits degrees of independence (or dependence) between invariants. Furthermore, we have provided practical tests for checking these notions in current software models, using a tool.

Finally, the expressions of operations $\wedge$ and $\vee$ on invariants in Table 1 were defined assuming complete independence between the combined invariants. Their definitions under the presence of some degree of dependency is ruled by equation (8) and is left as part of our future work, as well as the expression of further operations such as `implies`, `xor`, or logical equivalence.

6 Related Work

The idea of softening constraints is mostly used in the context of Constraint Satisfaction Problems (CSP), which are mathematical equations defined as a set of objects whose state must satisfy a number of constraints or limitations. In this domain, a soft constraint is a classical constraint where each instantiation of its variables has an associated value from a partially ordered set [BMR97]. When a CSP has parameters with interval values, it is called CSP with uncertainties [GN05]. Three main extensions of CSP permit softening constraints [Bar98]: *Fuzzy CSP* models constraints as fuzzy relations in which the satisfaction of a constraint is a continuous function of its variables' values, ranging from fully satisfied (1.0) to fully violated (0.0); *Probabilistic CSP* models those situations where each constraint has a certain probability to occur in the real system, hence allowing the modeler to reason about problems which are only partially known; finally, in *Weighted CSPs*, tuples come with an associated cost, which permits modeling optimization problems. Our approach could be seen similar to *Fuzzy CSP*, but in the context of OCL invariants and not as a CSP problem, and using Probability theory [dF17] instead of Fuzzy logic [Zim01].

A complete taxonomy of integrity constraints and their possible representations in OCL is described in Chapter 9 of [Oli07]. This taxonomy, together with the library of patterns for OCL expressions [AT07], provide interesting insights for characterizing different kinds of OCL invariants and how they can be softened.

The work [GHD18] presents different kinds of automated analysis of OCL invariants. In the current work we have covered invariant satisfaction and independence. Adapting the remaining six types of analyses to soft OCL invariants is part of our future research.

Our present work can also be considered as a generalization of our proposal for modeling occurrence uncertainty in models [BBMV18] using probabilities. In this paper, we have extended it to the case of OCL invariants, beyond objects and links. Our recent proposal on adding *belief uncertainty* to the UML elements of a model [BCC$^+$19], allows belief agents to assign a degree of belief to individual UML objects and statements. It deals however with a different type of uncertainty, hence providing an interesting complementary work to the one presented here.

Uncertainty has been considered in the UML context from various perspectives. For instance, the work [BMB$^+$18] shows how to extend the UML and OCL primitive datatypes with measurement uncertainty. The approach in [JPP$^+$08] concentrates on aspects of the modeling process of data, and UML models with explicit uncertain modeling of time are subject to [Gar08]. Imprecision in UML models is the topic of [SFP08]. The paper compares pros and cons of imprecision in the modeling languages UML and Modelica. Uncertainty has also been considered for model transformations. The work in [FSDC13] discusses how to handle transformations between models that incorporate uncertainty. Special emphasis on bidirectional transformation is handled

in [EPR15] and within the lenses framework in [DEPC16].

None of the mentioned approaches considers uncertainty from a fundamental modeling perspective in the sense that (a) the satisfaction of logical statements can be expressed in an uncertain way and (b) a prototypical implementation to experiment with uncertain constraints is provided.

Finally, other works propose extensions to OCL to account for further features, either as additions to the standard language, or in terms of libraries. Examples of the former include extensions for expressing temporal properties [DBB14, ZG03], system verification [ARAK17], or real time [LBJ15]. Extensions to OCL as libraries permit adding random operations [VG17], aggregation functions [CMPT10] or documentation and testing facilities [Chi09]. We initially considered developing a new, dedicated, extension to OCL, instead of staying with standard OCL. Such an extension would have probably resulted in a more readable, scalable and modular expression of the uncertainty in invariants. However, from our point of view, the benefits of using standard OCL and tools outweigh the disadvantages of deviating from it: our proposal is entirely backwards compatible with existing modeling languages, tools and methodologies, which can be used without any adaptation. This facilitates reusability, improves applicability, and significantly reduces the adoption costs of our proposal.

7 Conclusions and Future Work

This contribution discusses how OCL invariants can be relaxed to represent occurrence uncertainty in particular systems: we are interested in deciding whether a certain population of the system satisfies an invariant or not, but we are unsure about the actual occurrence of the elements of that population, and also about the degree of satisfaction that is actually required for the invariant to be fulfilled. A new satisfaction relationship has been defined, and we have shown how the new concepts and mechanisms have been prototypically implemented in practice, allowing system modelers to build system states and to reason about them.

The ideas presented in this paper can be continued in different directions. First, we would like to extend the initial list presented in Sect. 4.3 of OCL invariants 'patterns' that can be transformed into soft invariants, increasing the expressiveness of our approach. Second, we have currently studied satisfiability and independence of invariants. However, there are more validation use cases, as defined in [GHD18], which can be of interest for reasoning about the OCL specifications of the system. Third, more and larger case studies should give us more feedback on the features and expressiveness of our approach. In this sense, empirical validation and exercises with industrial modelers and users would help us validate its applicability, usability and effectiveness. We have proposed the use of `Dashboard` objects to represent the attributes that specify and regulate the level of flexibility of each invariant. However, the size of these objects can hinder their usability, particularly when dealing with large models that contain hundreds of invariants. Finding modular solutions to cope with this problem is another line of research we would like to explore next. Finally, we have initially focused on occurrence uncertainty. Extending our proposal to consider further kinds of uncertainties, in particular, belief and measurement uncertainty, is also left as part of our future work.

References

[ARAK17] Muhammad Waseem Anwar, Muhammad Rashid, Farooque Azam, and Muhammad Kashif. Model-based design verification for embedded systems through SVOCL: an OCL extension for systemverilog. *Design Autom. for Emb. Sys.*, 21(1):1–36, 2017. doi:10.1007/s10617-017-9182-z.

[AT07] Jörg Ackermann and Klaus Turowski. A library of OCL specification patterns for behavioral specification of software components. In *Proc. CAiSE'06*, volume 4001 of *LNCS*, pages 255–269. Springer, 2007. doi:10.1007/11767138_18.

[Bar98] Roman Barták. *Guide to Constraint Programming*, 1998. URL: https://kti.mff.cuni.cz/~bartak/constraints/extend_csp.html.

[BBMV18] Loli Burgueño, Manuel F. Bertoa, Nathalie Moreno, and Antonio Vallecillo. Expressing confidence in models and in model transformation elements. In *Proc. MODELS'18*, pages 57–66. ACM, 2018. doi:10.1145/3239372.

[BCC+19] Loli Burgueño, Robert Clarisó, Jordi Cabot, Sébastien Gerard, and Antonio Vallecillo. Belief uncertainty in software models. In *Proc. MISE'19*. ACM, 2019.

[BMB+18] Manuel F. Bertoa, Nathalie Moreno, Gala Barquero, Loli Burgueño, Javier Troya, and Antonio Vallecillo. Expressing measurement uncertainty in OCL/UML datatypes. In *Proc. ECMFA'18*, volume 10890 of *LNCS*, pages 46–62. Springer, 2018. doi:10.1007/978-3-319-92997-2_4.

[BMR97] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, March 1997. doi:10.1145/256303.256306.

[Chi09] Joanna Chimiak-Opoka. OCLLib, OCLUnit, OCLDoc: Pragmatic Extensions for the Object Constraint Language. In *Proc. of MODELS'09*, volume 5795 of *LNCS*, pages 665–669. Springer, 2009. doi:10.1007/978-3-642-04425-0_53.

[CMPT10] Jordi Cabot, Jose-Norberto Mazón, Jesús Pardillo, and Juan Trujillo. Specifying aggregation functions in multidimensional models with OCL. In *Proc. of ER'10*, volume 6412 of *LNCS*, pages 419–432. Springer, 2010. doi:10.1007/978-3-642-16373-9_30.

[DBB14] Wei Dou, Domenico Bianculli, and Lionel C. Briand. OCLR: A more expressive, pattern-based temporal extension of OCL. In *Proc. of ECMFA'14*, volume 8569 of *LNCS*, pages 51–66. Springer, 2014. doi:10.1007/978-3-319-09195-2_4.

[DEPC16] Zinovy Diskin, Romina Eramo, Alfonso Pierantonio, and Krzysztof Czarnecki. Incorporating uncertainty into bidirectional model transformations and their delta-lens formalization. In *Proc. of BX'16*, volume 1571 of *CEUR Proceedings*, pages 15–31. CEUR-WS.org, 2016. URL: http://ceur-ws.org/Vol-1571/paper_9.pdf.

[dF17] Bruno de Finetti. *Theory of Probability: A critical introductory treatment.* John Wiley & Sons, 2017. doi:10.1002/9781119286387.

[DP93] Didier Dubois and Henri Prade. Fuzzy sets and probability: Misunderstandings, bridges and gaps. In *Proc. of the IEEE Conf. on Fuzzy Systems*, pages 1059–1068. IEEE, 1993. doi:10.1109/FUZZY.1993.327367.

[Eck18] Peter Eckersley. Impossibility and uncertainty theorems in AI value alignment. Technical report, Partnership on AI & EFF, December 2018. URL: https://arxiv.org/pdf/1901.00064.pdf.

[EPR15] Romina Eramo, Alfonso Pierantonio, and Gianni Rosa. Managing uncertainty in bidirectional model transformations. In *Proc. of SLE'15*, pages 49–58. ACM, 2015. doi:10.1145/2814251.2814259.

[FSDC13] M. Famelis, R. Salay, A. Di Sandro, and M. Chechik. Transformation of models containing uncertainty. In *Proc. of MODELS'13*, volume 8107 of *LNCS*, pages 673–689. Springer, 2013. doi:10.1007/978-3-642-41533-3_41.

[Gar08] V. Garousi. Traffic-aware stress testing of distributed real-time systems based on UML models in the presence of time uncertainty. In *Proc. of ICST'08*, pages 92–101. IEEE, 2008. doi:10.1109/ICST.2008.7.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Comp. Programming*, 69:27–34, 2007. doi:10.1016/j.scico.2007.01.013.

[GHD18] Martin Gogolla, Frank Hilken, and Khanh-Hoang Doan. Achieving model quality through model validation, verification and exploration. *Computer Languages, Systems & Structures*, 54:474–511, 2018. doi:10.1016/j.cl.2017.10.001.

[GN05] Carlos Grandón and Bertrand Neveu. Using constraint programming for solving distance CSP with uncertainty. In *Proc. PPCP'05*, volume 3709 of *LNCS*, page 847. Springer, 2005. doi:10.1007/11564751_85.

[JCG08] JCGM 100:2008. *Evaluation of measurement data—Guide to the expression of uncertainty in measurement (GUM)*. Joint Com. for Guides in Metrology, 2008. http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf.

[JPP+08] Xiao Jing, Pierre Pinel, Lei Pi, Vincent Aranega, and Claude Baron. Modeling uncertain and imprecise information in process modeling with UML. In *Proc. of COMAD'08*, pages 237–240, 2008.

[Kos90] Bart Kosko. Fuzziness vs. Probability. *International Journal of General Systems*, 17(2–3):211–240, 1990. doi:10.1080/03081079008935108.

[Kow78] Robert A. Kowalski. Logic for data description. In *Proc. of Logic and Data Bases*, Advances in Data Base Theory, pages 77–103. Plemum Press, 1978. doi:10.1007/978-1-4684-3384-5_4.

[LBJ15] Aymen Louati, Kamel Barkaoui, and Chadlia Jerad. Temporal properties verification of real-time systems using UML/MARTE/OCL-RT. In *Formalisms for Reuse and Systems Integration*, pages 133–147. Springer, 2015. doi:10.1007/978-3-319-16577-6_6.

[Obj14] Object Management Group. *Object Constraint Language (OCL) Specification. Version 2.4*, February 2014. OMG Document formal/2014-02-03.

[Obj15] Object Management Group. *Unified Modeling Language (UML) Specification. Version 2.5*, March 2015. OMG document formal/2015-03-01.

[ODR+02] William Oberkampf, Sharon M. DeLand, Brian Rutherford, Kathleen V. Diegert, and Kenneth F. Alvin. Error and uncertainty in modeling and simulation. *Rel. Eng. & Sys. Safety*, 75(3):333–357, 2002. doi:10.1016/S0951-8320(01)00120-X.

[Oli07] Antoni Olive. *Conceptual Modeling of Information Systems*. Springer, 2007. doi:10.1007/978-3-540-39390-0.

[SFP08] Jörn Guy Süß, Peter Fritzson, and Adrian Pop. The impreciseness of UML and implications for ModelicaML. In *Proc. EOOLT'08*, pages 17–26. Linköping University Electronic Press, 2008.

[VG17] Antonio Vallecillo and Martin Gogolla. Adding random operations to OCL. In *Proc. of MoDeVVa'17*, volume 2019 of *CEUR Proceedings*, pages 324–328. CEUR-WS.org, 2017. URL: http://ceur-ws.org/Vol-2019/modevva_5.pdf.

[ZAY+17] Man Zhang, Shaukat Ali, Tao Yue, Roland Norgren, and Oscar Okariz. Uncertainty-wise cyber-physical system test modeling. *Software & Systems Modeling*, Jul 2017. doi:10.1007/s10270-017-0609-6.

[ZG03] Paul Ziemann and Martin Gogolla. OCL extended with temporal logic. In *Proc. of PSI'03*, volume 2890 of *LNCS*, pages 351–357. Springer, 2003. doi:10.1007/978-3-540-39866-0_35.

[Zim01] Hans-Jürgen Zimmermann. *Fuzzy Set Theory – and Its Applications*. Springer Science+Business Media, 2001. doi:10.1007/978-94-010-0646-0.

[ZSA+16] Man Zhang, Bran Selic, Shaukat Ali, Tao Yue, Oscar Okariz, and Roland Norgren. Understanding uncertainty in cyber-physical systems: A conceptual model. In *Proc. ECMFA'16*, volume 9764 of *LNCS*, pages 247–264. Springer, 2016. doi:10.1007/978-3-319-42061-5_16.

About the authors

Martin Gogolla is professor for Computer Science at University of Bremen, Germany and is the head of the Research Group Database Systems. In his group, foundational work on the semantics of and the tooling for UML, OCL and general modeling languages has been carried out. The group develops the OCL and UML tool USE (UML-based Specification Environment). Contact him at gogolla@uni-bremen.de.

Antonio Vallecillo is Professor of Software Engineering at the University of Málaga, Spain. His current research interests include Model-based Software Engineering, Open Distributed Processing, and Software Quality. Contact him at av@lcc.uma.es, or visit http://www.lcc.uma.es/~av.

Acknowledgments We should tank the anonymous reviewers for their insightful comments and suggestions, that have significantly helped to improve the paper. This project has been partially funded by Spanish Research Projects TIN2014-52034-R and PGC2018-094905-B-I00.